CONTECH: A SHARED MEMORY PARALLEL PROGRAM ANALYSIS FRAMEWORK

A Thesis
Presented to
The Academic Faculty

by

Phillip Vassenkov

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Computer Science

Georgia Institute of Technology
December 2013

CONTECH: A SHARED MEMORY PARALLEL PROGRAM ANALYSIS FRAMEWORK

Approved by:

Dr. Thomas M. Conte, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Umakishore Ramachandran
School of Computer Science
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: 11/11/2013

ACKNOWLEDGEMENTS

I would like to thank the TINKER group, at the Georgia Institute of Technology, for being supportive and guiding me through a process less foreign to them than to myself. They helped the Contech team in spotting flaws, contributing ideas, and finding kernels of ingenuity in our group discussions. I would also like to extend a special thank you to two TINKER members in particular, Eric Hein and Brian Railing, the major contributors of Contech. We worked together closely over the course of a couple years to design, plan, and implement the Contech framework as it exists today. Finally, I'd like to thank my adviser Tom Conte. His wisdom and encouragement has served me so greatly in my pursuit of knowledge and collegiate self-fulfillment that I find it difficult to succinctly put into words his impact on my academic career and passions in life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Page

SUMMARY

We are in the era of multicore machines, where we must exploit thread level parallelism for programs to run better, smarter, faster, and more efficiently. In order to increase instruction level parallelism, processors and compilers perform heavy dataflow analyses between instructions. However, there isn't much work done in the area of inter-thread dataflow analysis. In order to pave the way and find new ways to conserve resources across a variety of domains (i.e., execution speed, chip die area, power efficiency, and computational throughput), we propose a novel framework, termed *Contech*, to facilitate the analysis of multithreaded program in terms of its communication and execution patterns. We focus the scope on shared memory programs rather than message passing programs, since it is more difficult to analyze the communication and execution patterns for these programs. Discovering patterns of shared memory programs has the potential to allow general purpose computing machines to turn on or off architectural tricks according to application-specific features. Our design of Contech is modular in nature, so we can glean a large variety of information from an architecturally independent representation of the program under examination.

CHAPTER 1

INTRODUCTION

In the era of multicore computing, parallel programs have become more prevalent in order to exploit higher degrees of parallelism and decrease execution time. A parallel program is one that utilizes multiple processors simultaneously in order to perform a certain amount of computational work. By contrast, a serial program uses only one processing element to get that same amount of work done. Writing parallel programs requires programmers to factor in new levels of complexity not present in serial programs. Processors in a multicore machine (i.e., those with more than one processing element) each tend to operate as independent function units, unless otherwise instructed. Any cooperation among independently-executing processors requires them to exchange some sort of information. Orchestrating multiple processors to accomplish a common goal requires synchronization to guarantee program correctness.

Threads running on parallel processors executing a workload will spend a certain amount of time either performing computation or exchanging state information with threads on other processors. Some examples of shared state information may include pieces of shared data or ordering constraints among dependent pieces of computation. A very simple, classical model divides the time spent executing a program into processor computation and inter-processor communication. Performance bottlenecks in either domain could greatly reduce the efficiency of a parallel program, meaning poor resource utilization. Although computation and communication are constrained resources, communication costs are particularly high by comparison. This mostly stems from their

fundamental behaviors: computation is performed by rapidly executing instructions inside a processor, while communication entails sending data to other processor components, allowing them to carry out their purposes. A common theme in optimizations is to maximize efficient throughput across both computation and communication aspects of a program's behavior.

One example of a performance inefficiency (i.e., bottleneck) is an imbalance in the workloads across parallel threads. In an embarrassingly parallel program model, processors perform computation entirely in parallel with no communication. If one processor does more work than the others, the other processors remain idle while the busiest processor does the most work. The time spent waiting by the free processors could have been better spent performing the busiest processor's work, spreading out the workload and decreasing overall execution time.

Another source of inefficiency is redundancy, whether it is redundant communication or computation. For example, a coherence protocol in place to handle general coherency use cases may perform additional bookkeeping in a program where it is not necessary for a specific case. If there is a pattern of data sharing among processors with a specific characteristic, then a compiler armed with this information could force better use of the underlying memory coherence systems. Properly optimizing a parallel program is crucial to making the best use of available resources, consequently improving program performance further. Parallel programs come with new domains of optimization because of their need to express processor synchronization, which can include ordering constraints or memory consistency expectations.

Software is a means for expressing the intent of the programmer to the hardware. When a programmer writes a program, the source code describes what they hope the program to accomplish. After the source code is compiled, the executable is run on hardware. The hardware is tasked with interpreting the instructions and carrying out the programmer's intent. However, the type of hardware running the instructions can greatly impact program performance. The implementation details of the hardware dictate exactly how the hardware carries out the operations specified by the programmer. If an analysis of the program's runtime behavior were required, it would be helpful to only look at architecturally-independent information in order to reason about the original programmer's intent. Otherwise, the runtime behavior would be the programmer's intent as it was interpreted by a certain hardware implementation.

Understanding and controlling every aspect of the machine is infeasible to both the programmer and the software's scalability, so computer system designers build large-scale systems benefiting from the use of layers of abstraction. This layered approach enables a non-expert in a domain to utilize that domain's functionality without having to know the details. To illustrate, a biologist with some programming knowledge can solve problems in their domain without as thorough an understanding the details of a computer as that of a computer engineer. The more abstraction that exists between the programmer and the hardware behavior they desire, the more room there is for unintended hardware behavior side effects to crop up.

In general, a set of lower-level procedures may be composed together and exposed to a higher level as a function call to some "DoX()." The abstraction DoX() may entail allocating and initializing the resources for performing the operation *X*, executing

the operation, then freeing acquired resources, and initiating the necessary cleanup procedures. A prime example of this would be a lock. In the context of lower level operations, when acquiring the lock, a memory location is read into a register, modified, and written back into the memory location. The behavioral artifacts of acquiring and releasing a lock lie mainly in the caching subsystem and in the interconnect. These hardware side effects become especially apparent when observed in the large context of the entire program execution. Once apparent, they enable creating optimizations to leverage the side effects and boost performance. To illustrate, consider a highly contended critical section that is guarded by a lock. Inside of this critical section, the programmer unintentionally included thread local setup procedures required for the true critical section. If each thread of execution sets up its own data, then there may be extraneous lock contention where there is no need to protect those procedures in a critical section. The high degree of contention combined with the programmer erroneously making a critical section larger than it needs to be will harm the overall performance.

Current optimizations tend to try to remain as conservative as possible. Specific optimizations can offer a greater speedup by making more aggressive assumptions that would not be correct in the general case. Compiler optimizations, many having their roots in mathematical correctness proofs (e.g., dataflow analysis), must generalize to cover all possible program inputs. A compiler should only modify the program's behavior when it is apparent to the programmer that this optimization would not harm program correctness. A program compiled and optimized to be incorrect is of no use to anybody. A similar trend can be seen in processor design. Commercial processors are designed and built to support general purpose programs so that they may conservatively boost

performance in the general case. Eventually, the benefit to be gained from such an approach will dwindle. In the era of Dark Silicon [1], research is turning to heterogeneity and special case handling to get past this hurdle.

Most types of parallel programs need to communicate to have any meaningful use. There are two major types of parallel program communication paradigms: *shared memory* and *message passing*. Shared memory communication is where multiple threads of execution transmit data by reading and writing to memory locations in a common address space known to the multiple threads, typically on the same machine. In contrast, a different paradigm is message passing communication, which explicitly makes function calls to designate when a thread is receiving or sending data messages to another thread. Historically, as multicore processors and parallel program popularity began to bloom, so did the need for a means to pass data in between cooperating cores. A parallel program running on multiple cores within the same machine tends to communicate over shared memory due to its design simplicity. In contrast, when a program's resource demand exceeds the computational capabilities of one machine, a parallel program may run across a "supercomputer" comprised of multiple machines. Once past the bounds of a single machine, horizontal scaling concerns, in terms of performance and code maintainability, become more important. Although message passing has a slightly more complicated programming model, it begins to dominate in the inter-machine communication domain. There are many distinctions between shared memory and message passing communication, but one important difference is how synchronization and ordering constraints are enforced. Shared memory communication relies on locks, signals, and barriers to force specific orderings among threads executing parallel regions of code, but

message passing has these constructs implicit in the send and receive operations. A message receive operation can block further execution in the current thread until the corresponding send in another thread initiates a data transfer. This is a natural way to express "do not execute this code until the depended-upon computation has completed." A similar shared memory implementation would require a means for transmitting the data as well as enforcing the ordering constraints. Contech's focus is on shared memory communication, so message passing is outside the scope of this thesis.

When transitioning coding practices from serial programs to parallel programs, new issues crop up that did not exist before. In a correct parallel program, synchronization enforces ordering constraints where they are necessary. All other instructions' execution must only satisfy a partial order due to data dependences. Without synchronization, regions of code that require a certain ordering may be executed out of order and yield incorrect program results. If this undesirable scenario occurs in the context of data accesses, then it is referred to as a ***data race***. For example, if two threads try to increment the value of a shared variable, each must read the value, increment it, and write it back to memory. Ideally, one thread would increment the variable, followed by the second thread doing the same, resulting in the variable being incremented twice. Without proper synchronization, it is possible for the threads to read the value simultaneously, increment the variable, and both try to write the identical value back to memory. This is incorrect, as the variable should be logically incremented once per thread. To correct this issue, synchronization variables, such as locks, would be needed to ensure mutual exclusion while reading, incrementing, and writing back the value. When multiple threads employ synchronization communication, incorrect orderings on inter

thread synchronization can yield a halt in progress. A prime example of this is two threads, each trying to acquire two locks. The program would reach a state of *deadlock* if each thread acquires one lock and waits to acquire the missing second. Imagine a similar program, except instead of simply waiting to acquire both resources, on an acquisition failure, the thread will free all acquired resources thus far and try again to acquire both locks. The program would then be described as being in a state of *livelock*. The difference with respect to deadlock is the progressive change of program state, but a common attribute is the lack of overall forward progress.

When threads communicate, they need to pass information between each other. Since shared memory communication passes information by accessing globally known memory locations, the communication cost is in the cache and memory hierarchy, which keeps writes and reads coherent across various processor caches. Different memory hierarchy layouts could yield different low-level hardware performance artifacts. Another source of parallel programming concern is effective thread scheduling. The thread scheduler is the component responsible for granting access for a certain thread to execute on a processor. A naive thread scheduler would queue up freed resources and distribute them to threads on a first-come, first-served basis. A consequence of this approach is a pseudo-random assignment of threads to processors.

Consider a parallel program where all of the threads do not communicate in uniform densities. Smarter thread assignments could be made to minimize the number of hops traveled by messages on the underlying multicore processor coherency network. Amdahl's law provides a first order approximation of parallel program speedup with a finite number of processors. However, the generic definition assumes a homogenous and

balanced workload. During runtime, this is not always the case. If there is an imbalance in the amount of work that is distributed to each processor, the program will not finish execution until the thread with the longest running time completes. A more balanced workload distribution can yield faster execution time.

Contech is a framework created in order to analyze parallel program behavior, which is divorced from the architecture it runs on. Its efficiency and use of an architecture-agnostic program trace representation aids the Contech platform in evaluating future research ideas. Part of the framework is a compiler pass that instruments source code to record a trace for a particular run of a program. The other part of the framework is a set of libraries that allows analysis and evaluation of research ideas on the trace file produced. Contech captures program runtime behavior, which may be difficult to reproduce consistently, and does so with relatively low overhead. The analysis and evaluation tools in the framework were made with developers and scientists in mind, so they may conduct investigations with a low learning curve.

CHAPTER 2

RELATED WORK

There exist many program instrumentation and analysis frameworks. Their goal is to record information about the execution behavior of a program. The various frameworks tend to fall into one of several categories: *binary instrumentation, source instrumentation,* or *architectural simulation*. Portions of this chapter are also published as Georgia Institute of Technology Technical Report GT-CS-13-05 [2].

Binary Instrumentation

A binary instrumentation framework takes a compiled program binary as input and instruments the instructions. The instrumentation points can be decided at runtime, and instrumentation code is inserted appropriately. One such framework is Intel's Pin [3]. Pin is built as a Just In Time (JIT) compiler, responsible for instrumenting code and storing it for later execution in a code cache. Through many clever tricks, Pin achieves decent performance with light analyses, i.e. analyses that do not gather much information. Pin's design puts weight on the concept of transparency: the user should not need to modify their program in order to use a tool. Pin also stresses the importance of architectural-independence as it is "possible to write efficient and architecture-independent instrumentation tools, regardless of whether the instruction set is RISC, CISC, or VLIW." [3]

A similar framework to Pin is Valgrind [4]. Valgrind adopts a different design approach to program instrumentation than Pin. Users can write their own pluggable Valgrind tools that indicate where to instrument a program and how to analyze collected data. The Valgrind binary is then executed with the user tool and other parameters

specified via command line. The authors employ a heavier-weight instrumentation technique, tied closely to the idea of shadowing architectural state. They acknowledge that even lightweight tools, such as trace collectors, can perform poorly with this technique. Instead, they target resource-heavy analyses, since the same analysis could cost more had a cheaper instrumentation technique been chosen.

Helgrind [5] is a Valgrind tool that can help detect synchronization errors. The tool examines all memory accesses, as well as various synchronization events, and tries to establish which events happen before others. The *happens-before* relation was first formalized in Lamport's work [6]. Helgrind uses this relationship to find lock ordering issues, data races, or other incorrect uses of the Pthreads API. Helgrind+ [7] expands on Helgrind by improving the accuracy of the algorithm via locksets and by using different race detection models based on program execution length. False positives are an issue for race detectors and using different race detection models allows for different accuracy improving techniques to be applied.

A trend with binary instrumentation frameworks is that they perform analyses online with program execution. Pin authors acknowledge that online analysis can greatly impact program performance and behavior since, "the runtime overhead of executing analysis routines highly depends on their invocation frequency and their complexity." [3]

Source Code Instrumentation

Source code instrumentation is another possible method to record traces detailing program behavior. LLVM [8] is a popular open source, extensible compiler framework that allows users to write their own compiler passes and plug them into the regular compilation procedure. One way to use LLVM is to add instrumentation into the source

code before it is compiled into an executable. LLVM's robust API allows compiler pass writers to iterate over the abstract syntax tree representation of a program in order to insert instrumentation at specific points. When the instrumented binary is run, the code performing instrumentation is triggered by runtime events.

Architectural Simulation

Recording program behavior by means of architectural instrumentation is another possible design path. Architectural simulation leverages software flexibility in an attempt to observe how some model of a hardware architecture or microarchitecture behaves. Since there is a software model of a system, trace recording code can be inserted or run where necessary in a system simulator.

SimpleScalar [9] is an execution driven architectural simulator: the instructions to simulate come from a user binary rather than a trace. When running a program through an architectural simulator, richer information comes from an execution driven simulation than from trace driven simulation. Trace driven simulation is limited to the information gathered from one run of the program. While SimpleScalar attempts to balance performance, flexibility, and detail requirements, Contech has much more aggressive performance requirements.

Another very popular architectural simulator is gem5 [10]. The authors greatly stress the simulator's flexibility in catering to different users' needs. The simulator can leverage, on a per component basis, a tradeoff between accuracy and the level of simulation detail. Reduced levels of simulation detail will certainly boost performance, however the user will lose hardware artifact information once a detailed simulator component is replaced with a black box. Like Pin and many other tools, gem5 tries to

eliminate the decision between various methodologies, and instead tries to support any that the user may decide to employ. Expanding on the capabilities of a tool will attract more users and uses. Many architectural simulators, including gem5, serialize a user's parallel program because concurrently simulating parallel programs with an architectural simulator remains difficult. However, Manifold [11] attempts to do just that and is a parallel architectural simulator that runs on parallel hardware. Manifold components can be interchanged because of well-defined interfaces between various components. Also, Manifold puts emphasis on the user's ability to extend the framework by writing new components. Building a platform, with powerful support libraries, enables a low overhead way for users to experiment and cheaply implement their own specific requirements.

CHAPTER 3

ARCHITECTURE OF CONTECH

From a high level perspective, Contech has three key purposes: collecting runtime data, refining and aggregating the collected data, and enabling analyses on the data. There are several possible design architectures enabling different ways to accomplish these goals, each with their pros and cons. In subjectively analyzing these approaches, metrics were defined on which to evaluate them: cost of implementation, information flexibility and robustness, performance cost, and performance scalability. The cost of implementation describes how many man-hours would be required to implement a solution using a certain architecture to achieve desired functionality. This is generally the initial time investment that the tool designers must account for. The next metric is a combination of information collecting flexibility and information robustness. It attempts to qualify design rigidity, how sensitive the cost of implementation is to changes in functionality requirements (i.e., flexibility) and the quality of the information collected (i.e., robustness). Given that all of the approaches had certain overheads, the performance cost metric would gauge the cost of the user's source code executing and yielding analysis results and fruitful data. The final metric in evaluating an approach was performance scalability, that is, how much would the performance cost increase as user requirements became more complex. Together, these four metrics helped compare the various design approaches for the Contech system. Portions of this chapter are also published as Georgia Institute of Technology Technical Report GT-CS-13-05 [2].

One approach to meeting the system design goals is via architectural simulation. In general, architectural simulators consist of an instruction fetching frontend and an

instruction simulating backend. The complexity of a dynamic instruction fetching unit in most modern architectures along with the numerous amount of events that need to be simulated on the backend reduce this approach's ratings across both the cost of implementation and performance cost metrics. Contech agrees with SimpleScalar's execution driven simulation model because more information can be gleaned about the programmer's intent from the binary rather than a trace. SimpleScalar's performance sliding-scale bounds the simulation performance by three to four orders of magnitude slower than native execution [9].

In order to increase the performance of the architectural simulator, one must reduce the accuracy of simulation detail by replacing detailed architectural components with behavior-emulating black boxes. This loss is detail can greatly harm the detection of unknown hardware behavior side effects. In addition, it is also not very flexible, as new user requirements would require much simulator modification. However, gem5 tries to offer some type of flexibility in the simulator frontend by emphasizing the removal of concern to use one programming methodology over another and simply supports as many as possible. Performance scalability is not a strongpoint for this approach because increase in user requirements puts an exponential degree of strain on the rest of the system.

Another approach to meeting the design goals is via binary instrumentation. This entails running software that inserts event recording function calls among binary instructions at runtime. Intel's Pin is a popular binary instrumentation framework that could have been used as a means for implementing Contech. Contech concurs with Pin's emphasis on architectural independence. Like Pin, Contech also strives to eliminate the

user having to modify their program to use a tool. The Pin framework allows users to create *pintools* to plug into Intel's proprietary instrumentation framework with the intent of controlling how a binary is instrumented and what analysis is performed. This enables a low cost of implementation when collecting program runtime behavior. The framework allows users to specify at which granularity to instrument the binary: instruction, basic block, function level, and more. The Pin framework will then call the user's *pintool* functions at the appropriate times, performing the necessary information collection, aggregation, and analysis, however, Pin differs sharply in that Contech supports only offline analyses. The performance cost can be low, but tends to scale poorly, especially if analysis is done as part of an online algorithm [12]. There is a fair deal of flexibility with the type of information that Pin can collect, however, the more information that is exposed to the *pintool*, the more of a performance hit the program will take.

The third method examined was a compiler pass based approach to source code instrumentation. Generally, writing compiler passes has an enormous cost of implementation because of their correctness requirements and complexity. A compiler pass has the greatest degree of information flexibility and robustness because it is the component that sits between software and bare metal hardware; it has access to most aspects of the software and hardware. The performance cost overhead of compiler passes is already pretty low, but with the added benefit of iterative compiler optimization even lower overheads can be achieved. The performance scalability of this approach is quite competitive. The compiler pass would add functions to execute at specific points in the code. These functions do not affect each other's performance. By comparison to the architectural simulation approach, where additional complexity in one component can put

strain on all other parts of the system, a compiler based approach has a lower performance scalability rating because of the independence of the instrumented functions.

Table 1 – Summary of the tradeoffs amongst the approaches

	Architectural Simulation	Binary Instrumentation	Source Instrumentation
Cost of Implementation	* *	* *	* * *
Information Flexibility & Robustness	* *	*	*
Performance Cost	* * *	* *	*
Performance Scalability	* *	* *	*

After analyzing the various possible approaches in designing such a system, it was decided that the best approach to use was one closely tied to the compiler approach. Table 1 summarizes the qualitative design tradeoffs for each approach. One of the goals is to have the lowest performance cost, but that was a common problem in modern program analysis frameworks: users have to wait excruciatingly painful amounts of time in order to get data about the nature of program execution. Another goal was to get the most detailed and appropriate information to enable the most powerful analyses. Performance insensitivity to user requirements, defined as performance scalability, was also a goal. All of these can be bought with the high implementation cost that comes with a compiler based approach.

3.1 Workflow and Terminology

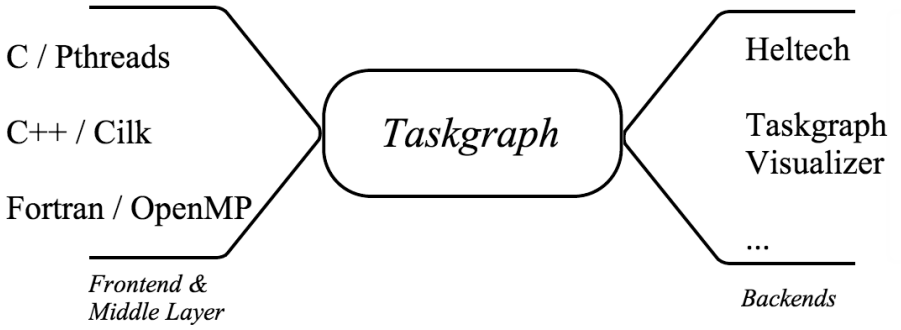

Figure 1 – High level workflow of Contech, starting from various types of source code, obtaining a task graph, and then performing analysis or transformation of it in the many backends of Contech.

Contech consists of a several stage workflow. Figure 1 offers a visual representation of the workflow [2]. First, the source file is compiled using the CLANG frontend in the LLVM framework, which includes the Contech compiler pass. The output of the compilation is an executable that contains additional instrumentation. When this executable is run, the computer runs the regular program code and the specially inserted instrumentation instructions as well. The instrumentation calls functions in the Contech runtime library, which results in the output of a trace, termed the *event file*. The event file is a partially ordered list of events, which describe different aspects of the program behavior. Since the event file is a raw dump of various events, the next step in the process is to refine this event file into a format that is more easily consumable after the program has finished executing. The event file is passed through a software converter, termed the *middle layer*, that aggregates and refines the events into another file known as the *task graph*. The task graph is a special file format that will be described in more detail below,

but it is in essence an aggregation of events that describe program behavior. The task graph file can then be fed into one of the many Contech backends, each of which facilitate data analysis or further data transformation. There was a great emphasis put on making these stages, and the interfaces between them, well defined in order to make the Contech framework modular.

In order to appeal to the broadest audience, a tool should support as many of the different programming framework as possible. Another reason the compiler approach made sense in this regard is due to compiler architecture in general. In order to assist with cross compilation and portability, compilers have front ends that can accept inputs from many different languages, that can all get refined down to an intermediate representation. A great example is one of LLVM's side projects that compiles Java bytecode into LLVM's intermediate representation [13]. The intermediate representation coming from any of the supported compiler frontends is an architecture independent assembly-like language. It can then be consumed by the compiler backend to generate machine code and an executable binary. The Contech architecture attempts to mimic this modularity. Since the LLVM-based Contech compiler pass instruments at the intermediate representation level of a source file, the many already existing compiler frontends can be used to support different languages in Contech.

Supporting different shared memory parallel programming models was another goal in the design of Contech. Currently, Contech only supports Pthreads; a user choosing to write their program in OpenMP, Cilk, or Intel's Thread Building Blocks would need to add their own instrumentation calls if they desired to use Contech. Fortunately, many of the programming models have constructs that function similarly.

Some examples of functionality common in many programming models are the ability to spawn threads, block certain threads from executing code (e.g. locks, barrier, sync, $private), or joining threads into another thread. The task graph captures program behavior in terms of synchronization primitives, so with some added modularity in Contech's source code instrumentation pass, the framework can not only support multiple languages, but also different shared memory parallel programming models.

3.2 Frontend

The Contech frontend is the section of the framework responsible for taking in source code, instrumenting source code with calls to the Contech runtime library, applying the standard compiler passes and optimizations, and producing an instrumented binary executable. After a survey of the various compiler frameworks available, LLVM seemed like a prime choice. It has a documented, well supported, and clean API that facilitates developers writing compiler components. LLVM's intermediate representation is also heavily documented, which is very helpful when extending a complex system like a compiler. One of the features of the LLVM framework is the ability to write a code transformation compiler pass (i.e., code generation, cross compilation, or otherwise), and register it as a callback in the LLVM compiler. Of the many granularities exposed to the programmer, the instrumentation begins at a higher level, the Module, and traverses deep enough in the computer stack to instrument individual instructions themselves.

Contech is concerned with collecting two "big picture" types of information common to all multicore architectures: memory access behavior and parallel execution ordering constraints. The event types of interest are gathered with the intent of making

Contech cross platform. Therefore, events representing common behavior or functionality regardless of programming paradigm or source language are instrumented.

Main

One of the first event types instrumented is the program's entry point or *main* function. The program's *main* is renamed and called from the actually run *main* method. The true *main* is located in the Contech runtime library, which is linked in during compilation. This new *main* function is responsible for performing the necessary instrumentation framework setup, such as allocating resources for components of the buffering architecture, and cleanup. This buffering architecture contributes heavily to a low overhead means for recording runtime events to a file. The *main* function is treated as a special type because it is the entry point into the program and Contech needs to leverage *main* to set itself up at a known point of entry. This eliminates the user's burden of having to insert special Contech initialization instructions into their code.

Parallel Programming Primitives

A majority of the other instrumentation event types are related to the synchronization primitives discussed earlier. When a program spawns a new software context (i.e., thread) this event is labeled as a *TASK_CREATE* event. Related to this event is the *TASK_JOIN* event that occurs when a program either terminates one of its threads or when a thread joins into another thread. These notions are common to any parallel program as it must have threads to execute in parallel serial sections of code that are created and destroyed. Another group of similar events are the actual synchronization primitives, namely *SYNC_AQUIRE*, *SYNC_RELEASE*, and *BARRIER_WAIT*. The two sync events, *SYNC_AQUIRE* and *SYNC_RELEASE*, correspond to the acquisition and

release of a synchronization variable such as a lock, a semaphore, or some other type of synchronization. Likewise, in order to capture multiple threads waiting on barrier synchronization primitives, barrier blocking operations must be captured. Capturing these parallel programming primitives is essential to displaying the ordering constraints amongst the various parallel portions of the program.

There is opportunity to exploit the reimplementation of synchronization primitives in different paradigms and allow the user to specify the event instrumentation triggers. In order to support different parallel programming paradigms, there is a decoupled mapping of an event type and a function name which triggers said event. The function map maps function names to the event types they represent. The function map serves as the entry point for adding event types throughout the rest of the Contech framework.

Memory Operations

Another essential aspect to portray in the program behavior, aside from the ordering constraints captured amongst parallel sections of code, is the memory access pattern of the program itself. There are a couple types of memory events that are instrumented: individual instructions and calls to memory transfer operations. Since Contech has access to LLVM's IR code, it can instrument individual load and store instructions, as well as function calls to memory transfer operations such as *malloc* or *free*.

Frontend Workflow

Initially the Contech compiler pass performs initialization and required setup such as setting up resources and getting runtime callback functions ready for instrumentation.

After setup has completed, the actual instrumentation may begin. LLVM's definition of a basic block allows for function calls inside of a basic block only if they return control into the same block, leaning closer to the definition of an extended basic block. All of the basic blocks must be normalized to have at most one function call per basic block, modifying the definition to be closer to the original definition of a basic block. Splitting basic blocks at function calls allows reasoning about per basic block memory behavior in a way that eliminates the possibility of outside basic blocks from influencing, i.e. loading or storing data into, external memory addresses.

Once the blocks are normalized, block level instrumentation may begin. This entails examining all of the instructions in a basic block and performing different actions based on the kind of instruction examined. Most of the actions require recording information. Writing out to file every time an event occurs would severely slow down the program. Instead, the data is recorded into an in-memory buffer that is managed by a special background thread. The general approach for recording information is to insert a function call into the LLVM intermediate representation to call code in the *ct_runtime* library, which in turn is linked into the binary.

Memory Access Behavior

Recording memory access information is crucial to understanding how data moves around in a program. In the event of any type of memory transaction operations encountered, whether it is a load, store, dynamic memory allocation, or memory transfer functions (e.g., *memmove*, *memcpy*, etc...), a callback to the *ct_runtime* library is inserted in order to record the memory operation. Storing the record in the buffer on every occurrence of a memory event allows Contech to gather the greatest amount of

information while keeping runtime overhead low. Examples of captured memory information include the addresses involved and transfer size of operations.

Ordering

Another aspect of program behavior common to any parallel execution architecture is the ordering that serial segments of code are executed. The remaining function calls instrumented by Contech are the sources of ordering constraints. In the event of a function call instruction, Contech uses the function map to identify the type of event encountered. This decouples functionality from source code constructs, allowing Contech to be extended onto other programming paradigms expressing similar core functionality.

Task Create

When a program starts running, the first program thread executes instructions from the program's *main*. Any additional threads of execution spawned from the initial or subsequently spawned threads must come from program instructions, either implicitly or explicitly inserted by the programmer. When such an instruction is encountered, it is labeled as a *TASK_CREATE* event. The function call that spawns a thread is replaced by a call to the *ct_runtime* library. The *ct_runtime* library function acts as a wrapper for thread spawning behavior. This enables Contech to perform the necessary internal setup for thread creation, which includes generating unique internal thread IDs, resolving timing information, preparing the buffering component for this thread, and then leveraging the buffering component to store the *TASK_CREATE* event.

When Contech reasons about threads of execution from a thread-level perspective, several issues become apparent that have low cost solutions in place.

Threads are scheduled to run on a core by the process dispatcher of an operating system. For a variety of reasons related to fairness, threads can have their execution preempted, state saved, and scheduled to run on a potentially different hardware core at a later time. The event file references events originating from certain threads, and, in order to consistently refer to the same thread, an ID maintained by the *ct_runtime* is assigned to each thread. This is termed the *contech_id*. Each event has the relevant *contech_id* recorded as well.

The event file, being a trace, contains timing information so that consumers can reason about the relative length of wall-clock time spent performing certain actions. The goal is to obtain timing information with as low of a cost as possible. To do this, Contech needed to poll a computer's notion of time. The cheapest way to find this information is through the time stamp counter, a counter for the number of cycles since machine reset. For fast access, each processor has its own counter. With dynamic frequency and voltage scaling being a popular means of throttling modern processors, the possibility of clock skew among thread timing information is very real. Another threat to timing accuracy occurs in the event that a thread is preempted and scheduled to run on a different core. In that case, the timing information for events inside a single Contech could be inaccurate. To deal with these issues, some design-changing attitudes about timing information were adopted. Low cost timing information can be inaccurate in the worst case, which is why Contech backends don't rely on the accuracy of timing information throughout the event file, rather on ordering information among synchronization primitives. Timing information was included in the case that a backend analysis is able to rely on potentially inaccurate timing information, such as querying the approximate number of cycles spent

in a thread or a critical section. This relaxed accuracy timing information is recorded based on the assumption that all cores have spent the same number of cycles since machine boot, at the same clock speed.

Task Join

At some point in a parallel program it may be desired to ensure that a thread has completed execution before progressing further in the program. One example of this behavior occurs towards the end of the program. Since it is usually undesirable for the main thread of execution to terminate itself and consequently all of its spawned threads, this mechanism enables the program to deal with mentioned problem. When one thread waits for the termination of another thread, this is referred to as a *TASK_JOIN* event. Such an event, when encountered, is forwarded to the buffering component for eventual output to the event file.

Sync Acquire

Managing critical sections and the number of threads allowed to execute some code is done usually through the use of synchronization variables. This includes barriers, but differences are to be discussed later. When Contech comes across a thread acquiring a synchronization variable, the type of event recorded will be *SYNC_ACQUIRE*.

Sync Release

The amount of time spent holding a synchronization variable while in a critical section can vary, so the event file must note not only the acquisition, but also the release of a lock variable. Sync variable release is noted in the event file with a *SYNC_RELEASE* event.

Barrier

When a program needs to have several threads all reach a section before proceeding forward, this is traditionally done with a barrier or similar structure. This event is marked as a *BARRIER_WAIT* in the event file. Although the frontend encounters a barrier as a single function call, the event file actually gets two events inserted. One event is for arrival at a barrier, while the other event designates barrier departure. With both events Contech can deal with issues that arise in the context of a loop embedded barrier. The task graph must distinguish between consecutive iterations of a loop reaching the same barrier. This can occur in the edge case arising when a thread is able to leave the barrier, jump to the beginning of the loop, and reach the barrier entry point again, before any other threads have left the barrier.

Buffering Architecture

As the compiled program runs, Contech inserted instrumentation is executed and events are written out to disk. The events written out to disk are in a certain order. There are two types of order preserved in the event file: partial ordering and semantic ordering. The partial ordering is present because events that are tagged with the same *contech_id* will come from the same thread of execution. Amongst all the events that originate from the same thread and share the same *contech_id*, there exists a total ordering. The semantic ordering refers to the assumption that the source code is correct and ensures events have a correct logical ordering between them, e.g. create before accompanying join, acquire and the matching release. Several approaches were evaluated in order to finally conclude on the actual implementation.

When evaluating the various possible approaches, the metric used to quantify the fitness of an approach was its overhead in terms throughput, overhead's impact on shared resources and the user program, and runtime footprint. If each instrumented callback to *ct_runtime* wrote out to file directly it would have a great effect on the user program and have entirely too much overhead. In order to reduce the impact of both factors, respectively, events should be written out to file asynchronously and they should also be buffered/aggregated for batch writing. Ideally, the events would all be collected with minimal overhead at runtime, and then written out to file after the user program completes execution. The issue with this approach is that the buffers would get too large. Without resorting to the performance decreasing swap-based solution, the program will crash once it fills the machine's main memory. To decrease the overhead of collecting events at runtime, Contech uses fixed size buffers and flushes them in a transactional fashion. Transmitting information primarily through the use of pointers further reduces the time spent copying data at runtime.

To summarize, it's a good idea to discuss the final outcome of these design decisions. A program instrumented by Contech will have function calls inserted at certain key points to record events. When any new threads are spawned, Contech allocates a buffer chunk in thread local storage. For most cases, when control is transferred from the user's program to the *ct_runtime* an event is copied into the buffer. Since these buffers are of a fixed size, a mechanism must exist to ensure that they do not overflow. Contech inserts buffer fullness checks at certain points, such as at the end of every basic block. In order to reduce realistically redundant checks, the Contech compiler pass utilizes a dominator tree to remove redundant basic block buffer size checks inside loops, placing a

check only at the loop header. When a buffer is detected as full, it is queued to be written out to file by a background thread. The user program thread then grabs an empty buffer from a pool of buffers. Once the background thread drains the full buffer, it is returned to the pool for reuse.

3.3 Middle Layer

While the event file is a complete trace of the events captured throughout program execution, it is not easily consumable in terms of information format. The next component in the Contech framework is termed the middle layer. It is the component responsible for aggregating events from the event file into discrete units of work that may proceed in parallel and finally connecting them with ordering constraints to form a task graph. The goal of this format is to be able to describe parallel program behavior in a succinct format, in a manner that does not tie it to any specific computer architecture.

A task is similar to a basic block, except at a much higher abstraction level. A task is a serial piece of code (i.e., series of basic blocks) that is terminated by one of the fundamental parallel programming constructs, such as acquiring or releasing a synchronization variable, or creating a new thread. A task is comprised of the code that runs on a thread. Each task contains information to answer three questions: what code ran, what memory addresses were accessed, and what other tasks depended on this one. The first two questions are answered by a list of chronologically ordered *ct_action* structures that describe a list of basic blocks executed or memory operations performed, either via a sequence of loads and stores or grouped into a memory transfer operation. The chronological ordering of events within a task is guaranteed because the events occur sequentially in one logical program thread.

Table 2 – Parallel programming synchronization primitives

Contech	Pthreads	Cilk	OpenMP
Create	pthread_create	Cilk_spawn Cilk_for	omp task omp for
Join	pthread_join	Cilk_sync	omp taskwait
Sync	pthread_mutex	Cilk_lock	omp critical
Barrier	pthread_barrier	(implicit)	(implicit)

The explicit information presented in the task graph besides a task's contents are the graph edges between tasks. These edges designate execution dependencies between tasks. The task graph has several special tasks, exemplified by table 2, which are distinct from the tasks discussed so far [2]. These special tasks correspond to the synchronization primitives discussed earlier, i.e. *TASK_CREATE*, *TASK_JOIN*, *SYNC_ACQUIRE*, *SYNC_RELEASE*, and *BARRIER_WAIT*. The purpose of having these special tasks is to be able reason about the successor and predecessor tasks separately from regular tasks. The special tasks form special ordering constraints among tasks. A simple example is *TASK_CREATE*: in the graph, a task entering a special *TASK_CREATE* task will then be split into two tasks. A more complicated example is the special task of *BARRIER_WAIT*. When a program contains a barrier for multiple threads, the corresponding tasks will have their successors be a single *BARRIER_WAIT* task, and the successors of the *BARRIER_WAIT* are all tasks that can then begin to execute once the necessary number of threads arrive at the barrier. There is a generalized statement that can be made about one or more tasks connected together: a task cannot begin execution until all of its predecessors have completed execution. One interesting corollary of this is that this property is transitive. Since a task graph is a partial ordering of tasks, a partial order's property of transitivity can be applied here. For some chain of tasks, it can safely be said

which tasks can happen before others. If A's successor is B and B's successor is C then A and C have a ordering between them. The goal in designing the task graph layout was to capture a program trace that would enable reasoning about which pieces of work can or cannot execute in parallel, while striving to be free from hardware specific information. Once a trace is collected, all the information necessary to analyze program behavior offline is read.

The middle layer receives as input the event file, as it was output by running the instrumented user program. During initialization, one key data structure allocated is a list of tasks per thread. Events are bucketized into the appropriate task list and coalesced to form tasks. Since the event list is a chronological partial ordering, this property is also carried forth into the task graph. After one of the synchronization primitives is encountered, it signals that all the events pertaining to a certain task have been processed and that a new task may be started. These special events take on special meaning in the middle layer as well. After a *TASK_CREATE* event is encountered, it brings up the question of whether the event belongs to the parent task or the newly spawned child task. This is answered via a boolean array describing which threads have been already executing, the parents, and which ones are only beginning to execute, the children. The two types of *SYNC* events, *SYNC_ACQUIRE* and *SYNC_RELEASE*, pose their own bookkeeping challenges as well. The middle layer tracks lock ownership, at the address granularity, in order to create the association between a lock variable being locked and unlocked by a thread. *BARRIER_WAIT* event types also have their own technical difficulties. In order to handle the possibility of a barrier being reached at a specific address multiple times throughout program execution, the middle layer maintains a list of

barrier coordination information amongst threads. The loop embedded barrier issue can arise because only a partial ordering of barrier events is guaranteed and every dynamic barrier instance corresponds to the same static barrier instance. It is guaranteed that all the arrivals into a dynamic instance of a barrier occur before any departures. No guarantee exists to prevent a thread's arrival at the next dynamic barrier instance before all threads have left the current barrier instance. Another one of the middle layer's responsibilities is recalculating the absolute clock cycle information to be relative to the start of the user program. Once the middle layer finishes processing through the event list, and constructing all of the tasks, they are then written out to disk. The bucketized nature of middle layer's algorithm allows for a deterministic ordering on the task graph output. Tasks are written to file chronologically, and in the event of an end time conflict among tasks, the tie-breaking rule is writing out in ascending *contech_id* order. The serialization of tasks is handled by another library, known as *taskLib*. Among other things, it facilitates reading and writing task graph files, and provides an API for interacting with individual tasks.

3.4 Backends

A Contech backend is modular piece of code, which consumes a task graph and performs transformation or analysis on it. The types of backends have been generalized to three different types, labeled by the backend's type of output: modified task graph, complex format, or statistics. A backend that outputs a modified task graph could potentially be modifying the structure or contents of the task graph in order to gain further insight about program behavior or reducing noise. Other types of backends, which output other complex formats, transform the task graph into another type of format, as long as they

describe program behavior to some degree. The third type of backend is one that produces statistics. These backends perform analysis on a task graph and output metrics to summarize what has been learned by the analysis. These backends are assisted by an API in *taskLib* which enables backends to read and write task graph files, along with accessing the different robust pools of information in a task, such as the streams of memory operations, basic blocks executed, or successors/predecessors of the task.

CHAPTER 4

RACE DETECTOR BACKEND EXPERIMENTS

4.1 Helgrind and Helgrind+

Data races are a potential danger for parallel program correctness. One of Valgrind's tools is a race detector called Helgrind. To demonstrate Contech's capabilities in terms of backend analysis robustness, a backend named Heltech was implemented to emulate Helgrind's behavior. Helgrind's race detection algorithm depends on establishing the happens-before relationship between certain pairs of events to determine whether or not conflicting data accesses pose a potential data race. The Heltech algorithm can achieve a decent level of accuracy compared to Helgrind. The algorithm processes a task graph in a breadth first fashion, keeping track of any memory accesses that can occur in parallel. If Heltech discovers two memory accesses that have the potential to occur in parallel based on the task graph's ordering constraints and one of them is a write, then a potential race is reported. The emphasis is put on writes because reads can occur in parallel safely with each other. The breadth first traversal of the algorithm is what ensures that dependent tasks do not get processed before anything that executes before those tasks. Heltech leverages task dependence transitivity to try and find a path in the task graph between the tasks encompassing the conflicting accesses in order to establish the happens-before relationship.

One of Contech's goals is to be a low implementation cost platform for research experimentation. A proof of concept implementation of an algorithm improving Helgrind's accuracy, termed Helgrind+, is used to demonstrate. One of Helgrind+'s main contributions has to do with improving accuracy by fusing in a lockset algorithm in a new

way. Once this concept is extended onto Heltech, a new backend is born: Heltech+. With the introduction of locksets, Heltech+ must keep track of which threads currently hold which locks. If a conflicting access is detected, the race is only reported when either thread holds no locks.

Quantification of race detector accuracies is made possible by a data race benchmark suite, termed *data-race-test* [14]. The benchmark suite consists of small C++ programs that reproduce many different types of data race conditions. Some of the tests are negative tests, tests that contain code that looks like a data race, yet proper synchronization exists. The rest of the tests are positive tests that contain a genuine data race condition, i.e. without proper synchronization. For each of the positive tests, it is sufficient to simply detect a race once before marking the test a success and moving on. For negative tests, it is not so simple because not detecting a race in one of these does not guarantee that a race does not exist. A computer contains a high degree of nondeterminism and this affects the programs that they run. A program contains many different partial-ordering constraints, such as thread scheduling, memory reordering, and out of order instruction execution. A single run of a program may not have exhibited a data race due to the ordering of that particular run, so multiple runs may be required to reveal the existence of a data race. The race may only become evident after certain conditions are met, so some work is required to try and recreate the necessary condition.

4.2 Contech Overheads

Contech's design went with a source code instrumentation approach, which involved a frontend written as an LLVM compiler pass. Many community experts often questioned why Intel's Pin was not used at the frontend to generate an event list, as it is

already a popular instrumentation and analysis framework that does not require recompilation of code. Also, why not simply use Pin itself to perform instrumentation and analysis? The arguments are that dynamic instrumentation, as well as online analysis, can impact program performance greatly. Contech is built around the concept of offline trace analysis and is not competing with dynamic binary instrumentation frameworks. To better investigate the tradeoffs, a Contech *pintool* entered development stages, with the goal of functionally mimicking the Contech LLVM frontend in order to produce an event list. Pin instruments callbacks to the *pintool* when certain Contech relevant events are encountered during binary execution. Although the Pin frontend is not yet fully developed, enough functionality exists to perform some overhead comparisons between Contech frontends. In its current stages, the Pin frontend has duplicated the memory access instrumentation and basic block instrumentation present in the LLVM Contech frontend. The proof of concept Pin based frontend will continue to be developed in the hopes of adding to Contech the feature of no longer requiring a user to recompile their program with the LLVM compiler pass.

In order to compare the instrumentation overhead of using one approach over another, the Pin frontend plugged into the same runtime as Contech with the purpose of writing out events to file using the same buffering architecture. In order to compare overheads of the approaches, both frontends were run with the Parsec 3.0 benchmark suite [15] and timing measurements were recorded. Due to the Pin frontend's proof of concept and developmental status, the entire suite has yet to be fully supported, so the focus was put on the Splash2 suite. The Splash2 suite was run with the *simmedium* input set and specifying 16 available cores. The system configuration is detailed in table 3. The

wall-clock times of native program, Contech instrumented program, and Pin instrumented program executions were recorded, and detailed in the next chapter, in order to compare Contech and Pin overheads, singling out the overheads of using each approach since they share the same runtime.

Table 3 – Experimental system configuration

Processor Model	Intel Xeon X5670
# of Processors	2
Cores per Processors	6
Hyperthreading	2-way
Clock Speed	2.93 GHz
Last Level Cache	12 MB
Main Memory	47 GB

CHAPTER 5

ANALYSIS AND DISCUSSION

5.1 Data Race Benchmark Suite Results

Once the race detector backends were implemented, Heltech, Heltech+, and Helgrind were run through the data race test in order to compare their accuracies. Tables 4 and 5 display the results and summarize the accuracies of the race detectors. Each cell contains a yes or no answer designating whether a race was detected or not. Since positive and negative tests have different success criteria, the tables have color coded red or green to aid in readability. Green cells contain expected satisfactory results, while the red cell's results are unsatisfactory.

The failed test cases were analyzed in order to potentially discover the sources of failure. With the causes narrowed down for most of the tests, some patterns emerged. A large number of the tests failed due to features unsupported by Contech. Since Contech looks at memory addresses as they travel through a program, synchronization information transmitted over some side channel circumvents race detection. Side channel synchronization techniques present in the data race benchmark suite include synchronization via file handles, or network sockets. Another reason for failures are features missing from Contech or Heltech/Heltech+. The race detector backends currently process memory accesses as a base address and length. Checking overlapping accesses is currently unsupported due to complexity concerns. Contech, also, does not support the thread local storage *__thread* keyword for C++. As these are not pressing research concerns, they have not yet reached the top of the list for future work.

However, despite all of the race test benchmark suite failures for Heltech and Heltech+, Helgrind also has a fair amount of failures. For the positive race cases, Helgrind achieves a failure rate of 0%, while Heltech and Heltech+ each fail 13.16% of the cases. However, Helgrind fails 50% of the negative test cases, while Heltech and Heltech+ fail 67.65% and 61.76% respectively.

5.2 Overhead Comparisons

In order to demonstrate why performance consideration led to a compiler based approach, the performance overheads of Contech's LLVM frontend and Pin frontend were pitted against each other. In order to compare the costs of binary instrumentation and source code instrumentation the two frontends use the same instrumentation runtime. Although having a task graph producing Pin frontend in the framework would be a useful asset, allowing Contech to support binaries as well as source code instrumentation, preliminary performance metrics indicate that if source code is available, it is better to use the LLVM frontend. Table 6 displays the overheads exhibited by the various frontends as they were run through the Splash2 benchmark suite. The Pin frontend averaged a slowdown of 59.70, compared to native executive. The LLVM frontend achieved a slowdown averaging to 50.83. If the slowdown between Pin and LLVM is calculated per benchmark, and then averaged, the average slowdown per benchmark is 6.72 The Pin frontend is already on average slower than the LLVM frontend with a subset of LLVM's functionality mimicked. The performance is not expected to increase if additional instrumentation is added due to Pin's poor performance scalability, as mentioned earlier.

There were a couple benchmarks, which had surprisingly results. Barnes and Radiosity had results that showed a speedup over the LLVM frontend. After further investigation, it became clear that both Barnes and Radiosity had a large number of locks [16] . Since the Pin frontend is still in development and locks were not yet supported, the LLVM frontend encountered the slowdown of instrumenting locks, but the Pin frontend did not.

Table 4 – Race detection tests for false race conditions

Test Name	Helgrind finds race?	Heltech finds race?	Heltech+ finds race?
NegativeTests.test11	N	Y	N
NegativeTests.test75	Y	Y	Y
NegativeTests.FileIOSynchronization	Y	Y	Y
NegativeTests.SocketIOSynchronization	Y	Y	Y
NegativeTests.PthreadOnceTest	Y	Y	Y
NegativeTests.test125	Y	Y	Y
NegativeTests.MmapTest	N	Y	Y
NegativeTests.MmapRegressionTest	N	Y	Y
NegativeTests.test141	Y	Y	Y
NegativeTests.CyclicBarrierTest	N	Y	Y
NegativeTests.Mmap84GTest	N	N	N
NegativeTests.epollTest	Y	Y	Y
NegativeTests.GetAddrInfoTest	Y	N	N
NegativeTests.Barrier	N	Y	Y
NegativeTests.StrlenAndFriends	N	N	N
NegativeTests.MemmoveTest	N	N	N
NegativeTests.StdStringDtorVsDtor	N	Y	N
NegativeTests.RunningOnValgrindTest	N	N	N
NegativeTests.BenignRaceInDtor	Y	Y	Y
NegativeTests.AnnotateIgnoreWritesTest	Y	Y	Y
NegativeTests.AnnotateIgnoreReadsTest	Y	Y	Y
NegativeTests.IgnoreBeginWithoutIgnoreEnd	N	N	N
NegativeTests.PublisherReader	Y	Y	Y
NegativeTests.PublisherAccessor	N	N	N
NegativeTests.PerThreadTest	Y	Y	Y
NegativeTests.StackReuseTest	N	Y	Y
NegativeTests.StackReuseWithFlushTest	N	Y	Y
NegativeTests.AtExitTest	Y	N	N
NegativeTests.EnableRaceDetectionTest	Y	Y	Y
NegativeTests.RepSanityTest	N	N	N
NegativeTests.RepNegativeTest	N	N	N
NegativeTests.BenignRaceTest	Y	Y	Y
NegativeTests.FlushVsJoin	Y	Y	Y
NegativeTests.LibcStringFunctions	N	N	N
Percent Failures	50%	67.65%	61.76%

Table 5 – Race detection tests for true race conditions

Test Name	Helgrind finds race?	Heltech finds race?	Heltech+ finds race?
PositiveTests.test110	Y	Y	Y
PositiveTests.test122	Y	Y	Y
PositiveTests.test146	Y	Y	Y
PositiveTests.CyclicBarrierTest	Y	Y	Y
PositiveTests.CyclicBarrierTwoCallsTest	Y	Y	Y
PositiveTests.LockThenNoLock	Y	Y	Y
PositiveTests.RWLockVsRWLockTest	Y	Y	Y
PositiveTests.MutexDtorNoSyncTest	Y	Y	Y
PositiveTests.MutexDtorNoSyncTest2	Y	Y	Y
PositiveTests.FprintfThreadCreateTest	Y	Y	Y
PositiveTests.HarmfulRaceInDtorB	Y	Y	Y
PositiveTests.HarmfulRaceInDtorA	Y	Y	Y
PositiveTests.AnnotateIgnoreReadsOnWriteTest	Y	Y	Y
PositiveTests.AnnotateIgnoreWritesOnReadTest	Y	Y	Y
PositiveTests.FalseNegativeOfFastModeTest	Y	Y	Y
PositiveTests.DoubleCheckedLocking1	Y	Y	Y
PositiveTests.DoubleCheckedLocking2	Y	Y	Y
PositiveTests.DifferentSizeAccessTest	Y	N	N
PositiveTests.RaceDetectedAfterJoin	Y	Y	Y
PositiveTests.RaceInMemcpy	Y	Y	Y
PositiveTests.RaceInMemmove	Y	Y	Y
PositiveTests.RaceInStrlen1	Y	Y	Y
PositiveTests.RaceInStrlen2	Y	Y	Y
PositiveTests.RaceInStrcpy	Y	Y	Y
PositiveTests.RaceInStrchr	Y	Y	Y
PositiveTests.RaceInStrchrnul	Y	Y	Y
PositiveTests.RaceInMemchr	Y	Y	Y
PositiveTests.RaceInStrrchr	Y	Y	Y
PositiveTests.RaceInStrcmp	Y	Y	Y
PositiveTests.RaceInStrncmp	Y	Y	Y
PositiveTests.ReadVsFree	Y	Y	Y
PositiveTests.FreeVsRead	Y	Y	Y
PositiveTests.RepPositive1Test	Y	N	N
PositiveTests.RepPositive2Test	Y	N	N
PositiveTests.RepPositive3Test	Y	N	N
PositiveTests.RepPositive4Test	Y	N	N
PositiveTests.FlushVsThreadStart	Y	Y	Y
PositiveTests.LibcStringFunctions	Y	Y	Y
Percent Failures	0%	13.16%	13.16%

Table 6 – Contech LLVM and Pin frontend overheads

Splash2 Benchmark Name	Pin Frontend Slowdown (over native runtime)	LLVM Frontend Slowdown (over native runtime)	Comparative Slowdown Factor (Pin/LLVM)
barnes	2.82	51.72	0.05
cholesky	32.73	31.31	1.05
fft	92.88	4.23	21.96
ocean_cp	29.43	17.79	1.65
ocean_ncp	25.87	12.97	1.99
radiosity	2.40	236.96	0.01
radix	254.00	9.72	26.13
water_spatial	37.45	41.96	0.89
Average	59.70	50.83	6.72

CHAPTER 6

FUTURE WORK AND CONCLUSION

Contech is by no means complete, and is still under active development. There are several engineering areas that may bear fruit in exploring.

The Contech frontend aims to support many languages and programming paradigms. With some additional engineering, Contech could leverage the modularity of its LLVM compiler frontend and support other languages such as Java, Fortran, or Objective-C. The same goes for new programming paradigms like OpenMP or Cilk. With these implemented, Contech would appeal to a much wider audience. More frontend work would entail bringing the Pin frontend to completion. The addition of this new frontend, despite performance overhead drawbacks, would broaden the audience further. Users would no longer be required to have the source code for the program they wish to analyze.

Another broad engineering area of future work has to do with data, particularly with reducing data created and data layout of the file formats passed between the various stages of the Contech framework. The event list contains the same events that would populate and reside in memory, waiting to be flushed onto the disk during runtime as events are created faster than they can be flushed. Reductions to the file size and average event size would also improve usage of shared resources, consequently reducing instrumented program runtime overhead. A usage pattern emerged in the task graph API: not all parts of the task graph were always used by an analysis. In order to reduce the amount of unnecessary data that is read for every task, some clever software engineering to selectively read data could reduce the average amount of time spent reading tasks from

disk. Another performance issue with the task graph format is the slowdown when attempting to access tasks randomly in a large file, which causes performance bottlenecks in the hard disk.

The last recommendation for further future work is for more backends. In order for users to get the most out of Contech, there should be more backends. Having more backends will not only increase the utility of the framework, but also set more examples and give more backend usage ideas to Contech backend developers around the world.

This thesis describes the design choices and inner mechanisms of the Contech parallel program analysis framework. Contech is a framework created in order to analyze parallel program behavior, which is separate from the architecture it runs on. Its efficiency and use of an architecture-agnostic program trace representation aids the Contech platform in evaluating future research ideas. Part of the framework is a compiler pass that instruments source code to record a trace for a particular run of a program. The other part of the framework is a set of libraries that allows analysis and evaluation of research ideas on the trace file produced. Contech captures program runtime behavior with relatively low overhead. The analysis and evaluation tools in the framework were made with developers and scientists in mind, so they may conduct investigations with a low learning curve.

REFERENCES

[1] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *Proceeding 38th Annu. Int. Symp. Comput. Archit. - ISCA '11*, p. 365, 2011.

[2] B. P. Railing, E. R. Hein, P. Vassenkov, and T. M. Conte, "Contech : A Tool for Analyzing Parallel Programs," Georgia Institute of Technology, GT-CS-13-05, Nov. 2013.

[3] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, and V. Janapa, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 190–200.

[4] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan Notices*, 2007.

[5] Valgrind-project, "Helgrind: a data-race detector," 2012. [Online]. Available: http://valgrind.org/docs/manual/hg-manual.html.

[6] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Communications of the ACM*, 1978, vol. 21, no. 7, pp. 558–565.

[7] A. Jannesari, V. Pankratius, and W. F. Tichy, "Helgrind+: An efficient dynamic race detector," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, 2009, pp. 1–13.

[8] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," *Int. Symp. Code Gener. Optim. 2004. CGO 2004.*, no. c, pp. 75–86.

[9] S. Modeling, "SimpleScalar : An Infrastructure for Computer Developed to provide an infrastructure for simulation and architectural," no. February, pp. 59–67, 2002.

[10] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. a. Wood, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, and T. Krishna, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1, Aug. 2011.

[11] J. Wang, J. Beu, S. Yalamanchili, and T. Conte, "Designing Configurable, Modifiable and Reusable Components for Simulation of Multicore Systems," *2012*

SC Companion High Perform. Comput. Netw. Storage Anal., pp. 472–476, Nov. 2012.

[12] M. Bach, M. Charney, and R. Cohn, "Analyzing parallel programs with pin," *Computer (Long. Beach. Calif).*, pp. 56–63, 2010.

[13] "LLVM Java Frontend," 2007. [Online]. Available: http://llvm.org/svn/llvm-project/java/.

[14] "data-race-test," 2012. [Online]. Available: https://code.google.com/p/data-race-test/.

[15] C. Bienia and K. Li, "Benchmarking Modern Multiprocessors," 2011.

[16] S. C. Woo, M. Ohara, and E. Torrie, "The SPLASH-2 Programs : Characterization and Methodological Considerations," no. June, 1995.